Detecting Code Clones in Binary Executables[*]

Andreas Sæbjørnsen Zhendong Su
University of California, Davis
{andsebjo, su}@ucdavis.edu

Jeremiah Willcock Thomas Panas Daniel Quinlan
Lawrence Livermore National Laboratory
{willcock2, panas2, dquinlan}@llnl.gov

Abstract

Large software projects contain significant code duplication, mainly due to copying and pasting code. Many techniques have been developed to identify duplicated code to enable applications such as refactoring, detecting bugs, and protecting intellectual property. Because source code is often unavailable, especially for third-party software, finding duplicated code in binaries becomes particularly important. However, existing techniques operate primarily on source code, and no effective tool exists for binaries.

In this paper, we describe the first practical clone detection algorithm for binary executables. Our algorithm extends an existing tree similarity framework based on clustering of characteristic vectors of labeled trees, with novel techniques to normalize assembly instructions, and to accurately and compactly model their structural information. We have implemented our technique and evaluated it on Windows XP system binaries with over 50 million assembly instructions. Results show that it is both scalable and precise: it analyzed Windows XP system binaries in a few hours and produced few false positives. We believe our technique is a practical, enabling technology for many applications dealing with binary code.

1. Introduction

Code duplication is common and hinders software maintenance, program comprehension, and software quality. *Clone detection*, the problem of identifying duplicated code, is thus an important problem and has been extensively studied. Many clone detection algorithms exist [2, 5, 7, 8, 10, 12], ranging from the basic string-based ones [2] to the more sophisticated ones based on program dependency graphs [5, 10].

Most existing clone detection algorithms operate only on source code, but not on binaries. However, the ability to detect binary clones is important because source code is not always available, for example, in the case of commercial off the shelf (COTS) software. A practical clone detection algorithm for binaries can enable many applications, such as the discovery of copyright infringements or the detection of sophisticated viruses, without requiring any source code.

Low-level binaries offer additional interesting challenges for clone detection. First, the problem demands better scalability because a single source statement is normally compiled down to many assembly instructions. Second, various choices made by a compiler, such as register and storage allocation, complicate detection. To see this, consider the following x86 assembly code:

```
mov [0x805b634], 0x0
mov [0x805b63c], eax
add esp, 0x10
mov eax, ebx
```

where `[0x805b634]` dereferences the memory location `0x805b634` (similarly for `[0x805b63c]`), and `eax`, `ebx`, and `esp` are registers. If we use the specific memory addresses or register names for clone detection, we will likely to be too specific to miss *true clones*. On the other hand, if we simply use opcodes (*i.e.*, mnemonics) of the instructions, we will likely to be too general and report *false clones*. Third, assembly instructions have a fixed, almost flat structure, while source programs can have arbitrarily deep structures. The rich structural information in source code is a key factor for clone detectors to perform well. All these differences require novel techniques for detecting binary clones.

In this paper, we present the first practical binary clone detection algorithm. Our algorithm follows a general tree similarity framework [7]: Instead of performing a quadratic number of pair-wise comparisons of instruction sequences, it models the essential structural information of the instruction sequences with numerical vectors and groups *similar* vectors to identify clones. We present novel techniques to generate precise and robust vectors for binaries and to compactly represent the vectors for improving scalability.

[*] This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and an LLNL LDRD subcontract. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by the Laboratory Directed Research and Development Program at LLNL under project tracking code 07-ERD-057. LLNL-PROC-406820

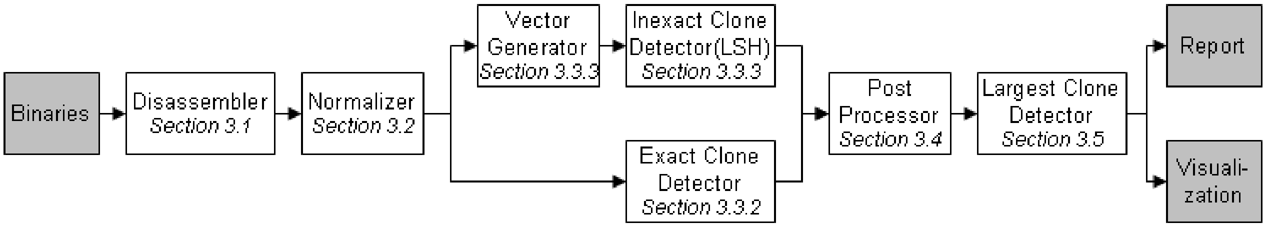

Figure 1. Disassembly and clone detection process.

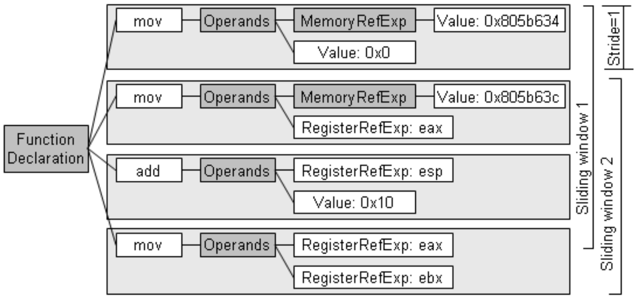

Figure 2. Example Syntax Tree for disassembled binary for stride 1 and window size 3.

We have implemented our algorithm and evaluated it on Windows XP system binaries with 50 million instructions. The results are promising and show that our technique is both scalable and precise. All the Windows XP binaries can be processed routinely under a few hours, and the detected clones are accurate with few false positives. Roughly 20% of the code appears in at least one clone cluster, which is consistent with results for source code [7, 8, 12]. The clone information also indicates certain modular structure in the code (see Section 5).

The rest of the paper is structured as follows. We first provide a high-level overview of our algorithm (Section 2). The detailed algorithm is presented in Section 3. Next, we discuss the implementation (Section 4) and evaluation (Section 5) of our algorithm. Finally, we survey related work (Section 6) and conclude (Section 7).

2. Overview

Figure 1 shows the flow-chart for our clone detection algorithm. This section explains this process with the help of the simple example from Section 1. Detailed technical descriptions of each step will be given in the corresponding sections shown in the figure.

First, we use a disassembler to process all input binaries and create their intermediate representations. For example, Figure 2 shows how we represent the sample instruction sequence from Section 1. Notice that each assembly instruction consists of a *mnemonic* (*e.g.*, `mov`) and an *operand list* (*e.g.*, `esp, 0x10`). Our intermediate representation preserves all binary file information, including instructions, header information, segments, etc; Section 3.1 describes the representation in more detail.

Second, the normalizer step (cf. Section 3.2) creates a *normalized* instruction sequence, abstracting away memory and register specific information. The following shows the normalized instruction sequence for the example:

```
mov MEM1, VAL1
mov MEM2, REG1
add REG2, VAL2
mov REG1, REG3
```

Third, we perform clone detection on the normalized instruction sequences. We separate the problem into two cases, mostly for efficiency reasons. One case is *exact clone detection*, where only identical normalized instruction sequences are returned. The other case is *inexact clone detection*, where certain differences are tolerated. This is a computationally challenging problem. We use *feature vectors* to approximate structural characteristics of the given assembly instruction sequences and group close vectors to find clones. See Sections 3.3.2 and 3.3.3 for more details.

We now have a set of *clone clusters*, *i.e.*, instruction sequences of a certain size[1] that are considered similar. For inexact clone detection, the similarity threshold is user defined while that between instruction sequences for exact clone detection is one. For instance, the clone cluster $C = \{seq_1, seq_2, seq_3\}$ contains three similar instruction sequences, seq_1, seq_2, and seq_3. Two sequences within a clone cluster, say seq_1 and seq_2, may overlap substantially and should not be considered clones. Removing such spurious clones is conducted by a postprocessing step (Section 3.4).

So far in the detection process, we consider only instruction sequences of a certain predefined length, but reporting many small clones is not as useful as reporting a few large ones. So, in the final step, we combine smaller contiguous

[1] This is referred to as the *window* as shown in Figure 2.

clones into larger ones. The algorithm for doing this is presented in Section 3.5.

3. Algorithm Description

This section gives a detailed description of our algorithm, structured according to the flow-chart in Figure 1.

3.1. Binary Disassembly

An assembly instruction is a pair of *mnemonic* m and a list of *operands* o. The mnemonic m represents the particular operation that the instruction performs, and is from a finite set M of possible mnemonics. The list of operands is a variable-length, but typically short, sequence of elements from the set O of possible operands. We partition the set O of operands into three categories: *memory references* (*e.g.*, `[0x805b634]`), register references (*e.g.*, `eax`), and constant values (*e.g.*, `0x10`). We do not use any of the structure of the individual operands other than this categorization, but we do assume the ability to compare two operands for syntactic equality. In the following algorithm descriptions, an instruction is defined as an element of the set $M \times O^*$, with O partitioned into O_{mem}, O_{reg}, and O_{val}. We use the functions $\mathit{mnemonic}$ and $\mathit{operands}$ to access the two parts of an instruction, and zero-based subscripts (of either single elements or intervals) to indicate accesses to elements or contiguous subsequences of a sequence; the $+\!\!+$ operator is used to indicate sequence concatenation, and the $+$ operator is used to add a single element to a set or bag. The function type maps from O to the set $\mathit{OPTYPE} = \{\mathit{MEM}, \mathit{REG}, \mathit{VAL}\}$ based on the particular category of an operand.

The full disassembly of a particular executable or library is defined as a sequence of functions, with each function containing a sequence of instructions. We define clones in terms of *code regions*, which are simply contiguous subsequences of the instructions of a single function, along with information on the starting address, function, and file of that list of instructions. We ignore that extra information for clone detection, but it is preserved by our algorithms and used by the postprocessing and visualization stages. We assume that algorithms that create and/or transform code regions implicitly process the extra information appropriately. When the distinction is important, the two functions $\mathit{instructions}$ and $\mathit{extraInfo}$ access the two parts of a code region. The actual process of creating the disassembled instructions for a program and grouping them into functions is implementation-specific; our particular implementation is explained in Section 4.

We split each function of a binary into code regions using two parameters *window* and *stride*. The window is the length of the code region to generate. The starting points of the code regions within a function are separated by the stride; note that code regions can, and almost always will, overlap. For example, with window size 50 and stride 10, the first three code regions contain instructions 0–49, 10–59, and 20–69 respectively. Algorithm 1 is used to compute the code regions within a particular function. As the window size and stride are constant for a particular run, this algorithm takes linear time in the number of vectors produced. For a single function F of length l, the number of vectors generated is $\lfloor (l - w + 1)/s \rfloor$.

Algorithm 1 Generate code regions

```
Require: $f$: Disassembled instructions for a single function, $w$: Window size, $s$: Stride
Ensure: $R$: The set of code regions
$R \leftarrow \varnothing$
for $i = 0$ to $\mathit{length}(f) - w + 1$ step $s$ do
    $\mathit{thisRegion} \leftarrow f_{[i, i+w)}$
    $R \leftarrow R + \mathit{thisRegion}$
```

3.2. Code Region Normalization

As explained in Section 2, the particular operands used in a code region may be specific to that code region, and so some "fuzziness" should be allowed in clones. For example, two regions may be identical except for certain constant values, offsets in memory locations, or particular addresses used as branch targets. In order to account for these differences, we normalize the instructions in a code region using Algorithm 2. This function takes a list of instructions in the $\langle \mathit{mnemonic}, \mathit{operands} \rangle$ format and converts them to be *abstract instructions* in the format $\langle \mathit{mnemonic}, \mathit{abstract\ operands} \rangle$. An *abstract operand* is a pair of an operand type (either MEM, REG, or VAL from the set OPTYPE) and a natural number indicating the index of the first occurrence of that particular operand expression within the code region. The operands are numbered separately for each operand type. The normalization produces an *abstract code region*, which is just a list of abstract instructions. The normalization algorithm takes linear time in the number of instructions in the code region, and is run once on each code region in the program.

3.3. Clone Detection

We define two ways to find clones among binaries: exact matching of normalized code regions, and inexact matching of feature vectors representing important aspects of the code regions. Both of these algorithms use linear time and space to find the initial set of clone clusters.

3.3.1 Definitions of Clone Pairs and Clusters

Code regions can appear in clone pairs and in clone clusters. A *clone pair* is an unordered pair of code regions that are "close enough" (by a metric defined later) to be considered to match. We form clone pairs into *clone clusters* by finding groups of clone pairs that all contain the same code

Algorithm 2 Normalize a code region

```
Require: r: Input code region
Ensure: r': Output abstract code region
  /* $N$ is a mapping from $OPTYPE$ to the sequence of operands of that type seen so far */
  $N \leftarrow \varnothing$
  $r' \leftarrow \langle\rangle$ with extra info $extraInfo(r)$
  for all instructions $i$ in $r$ do
      $ops' \leftarrow \langle\rangle$
      for all operands $o$ in $operands(i)$ do
          $t \leftarrow type(o)$
          if $o$ is an element of $N[t]$ then
              $idx \leftarrow$ zero-based index of $o$ in $N[t]$
          else
              $idx \leftarrow length(N[t])$
              $N[t] \leftarrow N[t] +\!\!+ \langle o\rangle$
          $o' \leftarrow \langle t, idx\rangle$
          $ops' \leftarrow ops' +\!\!+ \langle o'\rangle$
      $i' \leftarrow \langle mnemonic(i), ops'\rangle$
      $r' \leftarrow r' +\!\!+ \langle i'\rangle$
```

region. Always for exact matching, and in practice for inexact matching, the clone pair relation is transitive, and so choosing the neighbors of an arbitrary code region is appropriate.

For measuring the accuracy of our clone detection algorithm, we define false positives and false negatives. A clone pair is considered to be a *false positive* when it is found by the clone detection algorithm and yet the normalized instruction sequences of the two code regions in the pair are not identical. A clone pair is a *false negative* if it satisfies the definition of a clone pair given above and yet is not found by our algorithm. False positives and false negatives can never appear when using exact matching of normalized instruction sequences (by definition), but our inexact matching algorithm has both types of error. In order to test the accuracy of our algorithm, we test the inexact matching algorithm with a distance of less than one to simulate exact matching on feature vectors, and determine how well those results match the actual exact matching algorithm. Note that two distinct normalized instruction sequences may have exactly the same feature vector, so there can be false positives in a vector-based matching algorithm even when exact matches of vectors are found.

3.3.2 Exact Clone Detection

Exact matching uses a traditional hash table on the normalized instruction sequences, as shown in Algorithm 3. Although this algorithm produces a set of clone clusters, the corresponding set of clone pairs can be found by converting the partition C into an equivalence relation. This algorithm requires linear time, and produces exactly the correct set of clone clusters (that is, it does not have false positives or false negatives).

Algorithm 3 Find exact clone clusters

```
Require: $R$: A set of abstract code regions
Ensure: $C$: A set of clone clusters, each of which is a set of code regions
  $H \leftarrow$ empty hash table mapping from sequences of abstract instructions to sets of code regions
  for all code regions $r$ in $R$ do
      add $r$ to $H[instructions(r)]$
  $C \leftarrow \{c \in values(H) : |c| \geq 2\}$
```

3.3.3 Inexact Clone Detection

To find inexact clones, we adapt the basic approach developed by Jiang *et al.* [7] for locating source code clones. We characterize each code region using a set F of *features*, each of which identifies one property we consider important. For example, each possible instruction mnemonic is a feature, and each combination of the instruction's mnemonic and the type of the instruction's first operand is a feature. The features we use are local to each abstract instruction, and can thus be evaluated independently on the instructions in a code region. We count the number of occurrences of each feature within a code region, producing a *feature vector* for the region. Formally, a feature vector is a vector of natural numbers of length $|F|$, based on a fixed but arbitrary order of the features in F. We allow feature vectors to be indexed directly by features rather than requiring an explicit mapping from features to vector indices.

We count the following features of an abstract instruction or abstract code region; F is the disjoint union of the sets given. These five categories of features were necessary to consider for the feature vectors to accurately characterize the code:

- M, representing the mnemonic of the instruction;
- $M \times OPTYPE$, representing the combination of the mnemonic and the type of the first operand when one is present;
- $OPTYPE \times \mathbb{N}_k$, representing each normalized operand of the instruction whose index is under a chosen limit k;
- $OPTYPE \times OPTYPE$, representing the types of the first and second operands, in that order, of an instruction with at least two operands; and
- $OPTYPE$, representing the type of each operand in an instruction.

As we treat the window size as a constant, and the number of operands in an instruction is at most a small number,

Algorithm 4 *regionToVector*: Generate feature vector

```
Require: $r$: An abstract code region
Ensure: $v$: A feature vector
$b \leftarrow$ an empty bag (multiset) of features from $F$
for all instructions $i$ in $instructions(r)$ do
    $b \leftarrow b + mnemonic(i)$
    for all $\langle t, idx \rangle \in operands(i)$ do
        if $idx < k$ then
            $b \leftarrow b + \langle t, idx \rangle$
        $b \leftarrow b + \langle mnemonic(i), t \rangle$
        $b \leftarrow b + t$
    $ops \leftarrow operands(i)$
    if $length(ops) \geq 2$ then
        $b \leftarrow b + \langle type(ops_0), type(ops_1) \rangle$
$v \leftarrow bagToVector(b)$
```

Algorithm 5 Find inexact clones

```
Require: $R$: Set of abstract code regions
Require: $\delta$: Distance to use for queries
Require: $H$: Empty LSH hash table set
Ensure: $C$: Set of clone clusters
$V \leftarrow \{regionToVector(r) : r \in R\}$
for all vectors $v$ in $V$ do
    insert $v$ into $H$
$C \leftarrow \varnothing$
for all vectors $v_1$ in $V$ do
    if $v_1 \notin \bigcup C$ then
        $M \leftarrow vectorsInBucket(v_1)$
        $c \leftarrow \{v_2 \in M - \bigcup C : ||v_2 - v_1||_1 \leq \delta\}$
        $C \leftarrow C + c$
$C \leftarrow \{c \in C : |c| \geq 2\}$
```

we can treat the vector generation for a single code region as a constant-time operation. The overall vector generation for a large set of code regions is then a linear-time operation (each region can be processed independently). Algorithm 4 produces the feature vector for a code region. The $bagToVector$ creates a vector from a bag by counting the number of occurrences of each element of F in the bag.

Once code regions have been mapped to feature vectors, we then define the distance between two code regions as the l_1 distance between their corresponding feature vectors. The distance between the vectors is intended to approximate the dissimilarity between the code regions. We then define an inexact clone pair for a distance δ as an unordered pair of code regions $\{r_1, r_2\}$, with feature vectors v_1 and v_2 respectively, where $||v_1 - v_2||_1 \leq \delta$. The parameter δ affects the similarity required between the feature vectors: having $\delta = 0$ requires the vectors to be identical and thus the code regions to be almost the same, while larger distances allow more dissimilar code regions to appear in clone pairs.

Given the space of vectors and a distance metric, we would like an efficient way to find all vectors within a given distance from a query vector; i.e., we would like a *randomized near neighbor* data structure and algorithm. We follow the approach in Deckard [7] and use *locality-sensitive hashing (LSH)* for this purpose. In particular, we use the set of hash functions from [6] for the l_1 distance on vectors of natural numbers. LSH is an approximate algorithm, allowing false negatives in order to achieve constant time and space insertion and queries for distance-based matching when given appropriate parameter choices. Our inexact clone detection approach is shown in Algorithm 5. This algorithm requires an empty hash table set to be passed as its input; LSH uses two parameters to create the hash function, and they must be chosen carefully for good performance and accuracy. See Section 4.2 for more information. The function $vectorsInBucket(v_1)$ used in the code finds all vectors that hash into the same bucket as v_1 in any of the hash tables in the LSH hash table structure; this set is the same as the set of elements that would be searched in a near neighbor query for the element v_1 in H. Note that this algorithm produces clusters greedily in such a way that each code region is in at most one cluster. Although the order of iteration through the regions can change the set of clusters produced for non-transitive neighbor relations, we assume, supported by past literature, that the relation is transitive or close to it. Allowing overlapping clone clusters can lead to an algorithm requiring quadratic time, while the limitation to non-overlapping clusters uses only linear time.

Although Deckard [7] uses Euclidean (l_2) distances to detect inexact clones in source code, we chose to use l_1 distances instead: our data sets are very different from those used by Deckard and required us to recompute parameters for every group, which could be done analytically for the l_1 norm more easily than for l_2. According to Gionis *et al.* [6], the l_1 norm is a tighter bound than the l_2 norm.

Every inexact clone detection phase is optimized by first performing exact clone detection and thereby making sure that every distinct vector is only represented once when doing LSH-based inexact clone detection, even if that same vector is used for several code regions. The results of the exact and inexact clone detection are merged when both runs are done.

3.4. Removing Trivial Clones

When the stride s used to generate code regions is smaller than the window size w (i.e., the length of each code region), it is possible that two code regions in the same function almost completely overlap with each other. As the feature vector generation is almost independent of the order of the instructions, it is likely that these two regions would be detected as a clone pair. However, such a pair is not interesting as it is effectively stating that a region of

Algorithm 6 Remove trivial clones

```
Require: $C$: Set of clone clusters
Require: $o$: Allowed fraction of overlap
Require: $w$: Window size
Ensure: $C'$: Post-processed set of clone clusters
$C' \leftarrow \varnothing$
for all clusters $c$ in $C$ do
    $c' \leftarrow \varnothing$
    for all functions $f$ containing regions in $c$ do
        $R \leftarrow$ code regions from $f$ in $c$
        sort $R$ by instruction offset within $f$
        $\mathit{first} \leftarrow \top$
        $\mathit{lastOffset} \leftarrow 0$
        for all code regions $r$ in $R$ do
            $\mathit{offset} \leftarrow$ offset of $r$ within $f$
            if $\mathit{first}$ or $\mathit{offset} \geq \mathit{lastOffset} + o \cdot w$ then
                $c' \leftarrow c' \cup \{r\}$
                $\mathit{lastOffset} \leftarrow \mathit{offset}$
            $\mathit{first} \leftarrow \bot$
    if $|c'| \geq 2$ then
        $C' \leftarrow C' + c'$
```

code is a clone of itself. We define a parameter o that indicates the fraction of instructions that two regions can have in common and still be considered a legitimate clone pair, and reduce the set of clone clusters using Algorithm 6.

It is unclear what it means when two overlapping code regions r and r' are considered clones. In our experiments we have decided on an overlap of 50 percent or less. Although a few of the clones that are not filtered out can be considered false clones, we are more concerned with completeness of our clone-set than this problem. If it is desirable to do so, filtering out all overlapping regions can be done just by changing o.

This algorithm is finding the maximum independent set of an interval graph (the graph of overlapping vectors within a single clone cluster and function), a problem that can be solved using sorting in $O(n \log n)$ time and $O(n)$ space. Here, the nonlinear term is only applied to those code regions that are both within the same clone cluster and the same function, making the algorithm effectively linear.

3.5. Finding Largest Clones

Finding the largest sequences of instructions A and B that are part of a clone pair $\{A, B\}$ is important to avoid an overestimate in the reported number of clones. Since there are overlapping vectors A and A' in the set of vectors in C, we must expect that the number of clones and the total number of vectors in the clones is an overestimate. For instance, if there is a sequence of length n ($n \geq w$) that matches between functions f_1 and f_2 then it will be represented by up to $\lfloor (n-w)/s \rfloor + 1$ clusters in C. If n is 500, the window size w is 40, and the stride s is 1, that single logical clone pair becomes 461 pairs in the output.

Algorithm 7 Estimating the Largest Clone Pairs

```
Require: $L$: Sequence of clone pairs
Ensure: $L$: Set of largest clone pairs
sort $L$ using the order in the text
$n \leftarrow \varnothing$
$L' \leftarrow \{\varnothing\}$
for all $n'$ in $L$ do
    if $\mathit{overlap}(n, n')$ then
        $n \leftarrow \mathit{join}(n, n')$
    else
        $L' \leftarrow L' + n$
        $n \leftarrow n'$
$L \leftarrow (L' + n) - \{\varnothing\}$
```

The algorithm in Algorithm 7 finds the largest clone pairs $\{a, b\}$, but does not find the largest clone clusters. Clone pairs can be generated from a clone cluster by taking all unordered pairs of distinct code regions from the cluster. There may be a quadratic number of clone pairs from a single clone cluster, although clusters are typically not large. The algorithm relies on a total order among code regions; the code regions are first grouped by the functions containing them, and are sorted by the instruction offset within each function. The algorithm assumes that a is less than b in each clone pair. We sort the clone pairs according to the two elements of the pair in that order. Sorting ensures that if there are two sequences in A and B that overlap then all clone pairs for those sequences will be adjacent in the list.

Given a sorted list we do a linear search over the list of clone pairs where clones in sequence that overlaps are joined into a bigger clone pair. We define two clone pairs $\{a, b\}$ and $\{a', b'\}$ to be overlapping if the code sequences corresponding to a overlaps with a' and similarly b overlaps with b'. A sequence X then overlaps with X' if both sequences are within the same function f and they contain some of the same instructions.

When joined the two clone pairs are then replaced by a larger, joined clone pair. The joining is inherently optimistic and assumes that if two overlapping clone pairs are joined then the joined pair is also a clone. We expect this to create false positives, but since the number of those larger clones is small it is cheap to run a more expensive algorithm on the joined pairs in order to remove false positives.

The computational complexity of Algorithm 7 is $O(n \log n)$ where n is the number of clone pairs, as sorting is the most computationally complex task. The number of clone pairs is itself $O(m^2 N)$ in the worst case, where N is the number of clone clusters, and m is the size of the largest clone cluster. The actual number, however, is proportional to the sum of the squares of the clone cluster sizes, and the number of large clone clusters is not expected to be large.

4. Implementation

We implemented our algorithm as a clone detection tool that uses IDA Pro for disassembly and is therefore capable of interpreting both ELF (Linux) and PE (Windows) executable formats. Although we selected IDA Pro as a frontend for disassembling the binaries in this study, our implementation does not rely on it. We also use IDA Pro to recognize the functions in each binary. As part of the ROSE compiler project we have developed an open source disassembler; it has been used for clone detection on approximately a thousand Windows PE and Linux ELF formatted files but results from this work are too recent for inclusion in this paper.

Our clone detection and analysis system is implemented using C++ and uses a SQLite (version 3) database for communication. Each phase of the analysis is implemented as a separate program, allowing new analyses to be added modularly. The first pass converts a set of disassembled functions and instructions from IDA Pro into the ROSE intermediate representation, and from there to both feature vectors and normalized assembly instructions. These are inserted into the database. Given these, either exact or inexact clone detection may be run, each producing a new table of clone clusters; this table may also have trivial clusters removed during the clone detection process. Largest clones can then be found if desired, and then visualization can be applied to the resulting database.

4.1. Memory and Computational Efficiency

The dimensionality of the feature vector is 26 times larger for binaries than reported for source code in [7]. The memory usage and computational complexity of LSH thus increase by at least 26 times when using LSH on object code as compared to source code. Since each dimension takes at least one byte, and often more, it is necessary to create a compressed representation for large data sets.

We made the observation that our feature vectors are sparse and largely consist of small numbers. For example, we define a large set of features F that includes features such as the number of references to the 80th memory reference in a code region; it is unlikely that there are 80 memory references in a region, and so this element of the feature vector is almost always zero. Since we only generate the data set once and use it many times, it is beneficial to construct a compressed representation once and reuse it, saving disk and memory space. Because zero elements in the vector are handled specially, they can be skipped in some computations to save CPU time.

We use a compressed representation that run-length encodes contiguous sequences of zero elements in the vector, plus encodes numbers using variable numbers of bytes based on the values of the particular entries. Several contiguous vector elements that are each near zero can also be packed into a single byte. Experimentally, we have shown that this technique can use 17 to 36 times less space to store the same set of vectors. Generally, computation time is traded for memory when using compressed representations, but we were able to operate on the compressed vectors directly and thus take advantage of the fact that many vector elements are always zero to save computation in the vector kernels used by LSH (dot products, element extraction, norm computation, etc.), as well as not requiring time or storage for decompression. We store vectors in compressed form both on disk and in memory. Without compressing the vectors, our data sets would require hundreds of gigabytes of disk space; compression allows the same data sets to be processed entirely in memory if desired.

4.2. LSH Parameter Tuning

If a family of LSH hash functions H is used to find clones in $O(n \log n)$ time then the parameters controlling the probabilities of two similar elements colliding must be carefully chosen. The number of hash tables l and the number of elements in a single hash function k determine the runtime and accuracy of the algorithm. The general rule is that a larger k increases the false negative rate while a larger l increases the collision rate (i.e., the percentage of elements scanned in the query but that are not within the desired distance). An LSH data structure consists of l independent hash tables, each containing the same data but a different hash function; each hash function is built from k components. LSH's memory usage is thus proportional to l, even with the number of buckets and bucket size fixed. The k parameter does not affect memory usage substantially, and has only a weak impact on the time used for hash table operations; however, the value of k determines how many results are returned for a given query, leading either to unnecessary, failing distance tests or false negatives.

Parameter selection was challenging for our dataset since the dimensionality of our dataset is 26 times larger than in [7]. Our dataset has large distances between different vectors, and in particular the l_1 norms of the vectors in our data set vary. We observed through experiments that LSH does not handle such data sets well, and thus we apply LSH separately to sets of vectors grouped using their l_1 norms. The Deckard [7] system also uses this approach. Groups are chosen to overlap such that any two vectors that are within the distance bound δ are in at least one group together. We then choose the LSH parameters for each group separately. The experimental approach for selecting parameters as done in [1] was not viable for our application, and so we choose optimal parameters analytically.

We use the approach from [16] for this purpose. We use their model of LSH behavior to define a function from k, l, and the distance d between two vectors to find the probability of one being found in a query for the other. Assuming a uniform distribution of distances between vectors, we add the probability that vectors within the distance bound will not be found (false negatives) and the probability that vectors outside it will be found (collision rate). This sum provides a score for that particular set of parameters. We then use the cost model from [16] to estimate the time used for the given set of parameters, and vary k to optimize the cost for a given level of accuracy. We find l using a formula given in that paper; we can choose l to achieve an arbitrarily low false negative rate.

4.3. Experimental Setup

We performed a large scale study on our clone detection tool using the disassembled representations of the Windows XP sytem executables and libraries. All runs were done on a workstation with two Xeon X5355 2.66 GHz quad-core processors and 16 GB of RAM. We only use one of the cores in our experiments. We have a 4-disk RAID with 15,000 RPM 300 GB disks. The machine runs Red Hat Enterprise Linux 4 with kernel version 2.6.9-78. None of our runs used more than 4 GB of memory for inexact clone detection.

5. Experimental Results

In this section we evaluate our tool using a large scale study on the Windows XP system libraries for various window sizes. Accoring to Table 1 there are many small functions that are not covered by the larger window sizes. About 2/3 of the 1,108,535 functions in Windows XP system files are less than the smallest window size. These could be covered by the clone detection by reducing the window size. For window sizes 40 to 200 the number of files where at least one function is covered do not change considerably.

Table 1. Functions and files with at least one vector

Stride	# of files	# of functions
500	863	7,072
200	1,486	42,819
120	1,633	97,588
80	1,681	168,224
40	1,722	342,874

5.1. Clone Quantity

We explored the problem size for a range of different window sizes, total number of vectors, and clusters in our data set; see Table 2. We evaluatated the clone quantity relative to the percentage of the original code that any vector (found in the dataset) covers. If an instruction is covered more than once by different clones it is only counted once.

Table 2. Clone statistics

Window Size	Vectors	Clusters	Clones
500	2,588,507	206,785	704,263
200	7,963,384	587,582	2,039,093
120	13,130,524	966,604	3,419,038
80	18,304,493	382,023	1,227,669
40	27,946,044	2,368,355	8,636,593

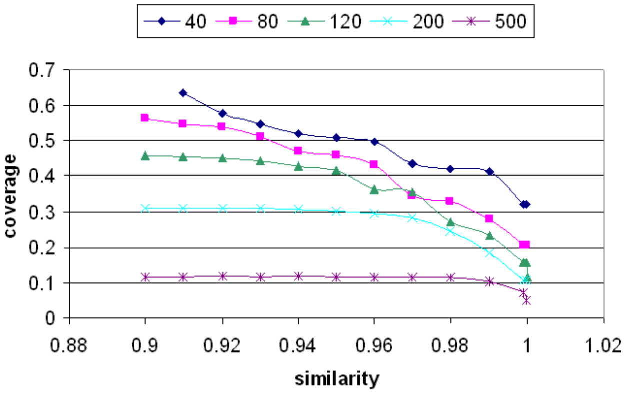

Figure 3. Total coverage

Figure 3 shows that the code covered increases with decreasing similarity for all window sizes, but the total coverage is much larger for smaller window sizes. The total coverage decrease with increasing window size because many smaller functions do not contain enough instructions to fill one sliding window since we respect function boundaries. When the similarity is decreased the covered code approaches the percentile of the code covered by the windows. For instance the coverage is 11.9% for window size 500 and 32% for window size 200. We also notice that smaller window sizes reach a higher level of coverage for a higher similarity grade. This is both due to instruction sequences of smaller window sizes covering smaller functions, and that smaller regions of the code may be clones even if they are contained within larger, non-cloned contexts.

Table 3. Size of Clone Clusters

Stride	> 2	> 4	> 16	> 64	> 128
500	69,775	16,705	2,420	531	0
200	196,540	59,336	5,694	905	11
120	325,010	105,773	10,007	1,404	125
80	467,737	157,983	15,442	2,117	337
40	798,272	286,066	32,585	3,919	890

Table 3 shows that when the window sizes increase the average size of the clone clusters decrease. This is a validation of the intuition that larger instruction sequences are more unique than shorter instruction sequences.

5.2. Clone Quality

Table 4. False positive rate using mnemonics

Window Size	False Positive Rate
500	15.54
200	16.92
120	18.31
80	19.19
40	26.08

Table 5. False positives and coverage

Window Size	False Positives (%)	Coverage (%)
500	3.3	3.0
200	2.9	7.9
120	2.9	12.5
80	2.9	17.1
40	3.2	27.8

In order to evaluate the clone quality we compare the results found using the normalized instruction sequence with the clones found using exact clone detection on the normalized instruction sequence. We define a false positive as any clone that is not a clone in terms of the instruction sequence, but which is still part of the clones found by LSH. This metric was used as a way to evaluate how well the feature vectors characterize the normalized instruction sequence when doing exact clone detection. Manual inspection of the clones found by our inexact clone detection found similar sequences of instructions for large window sizes, but the results were less reliable for smaller window sizes.

We found our false positive rate to be low as shown in Table 5. When only using mnemonics for detecting code clones the false positive rate is high as shown in Table 4

5.3. Scalability

Table 6. Vector generation time (min)

Window Size	VecGen
500	225
200	247
120	254
80	275
40	277

We show that our tool is scalable by doing a large scale study on the Windows XP system files for similarities between 0.90 and 1.0, where 1.0 is exact clone detection. For all the window sizes used in this study the stride is 1. This is the first research study that we are aware of which evaluates LSH on this large of a data set.

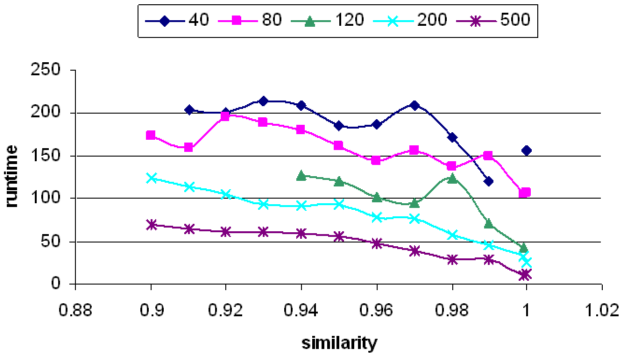

Figure 4. Runtime (in min)

The vectors are stored in a database for each stride and window size. This database is generated once and re-used later. Table 6 shows that we generate vectors scalably.

Figure 4 shows that our tool detects clones scalable on our vector database for all window sizes. The runtime seems to increase as expected for an $O(n \log n)$ algorithm with the two factors which contribute to an increase in the data set size; decreasing window size leading to more vectors and decreasing similarity grade leading to a higher likelihood of hashing vectors into the same bucket.

5.4. Estimation of Largest Clone Size

Table 7. Largest clone pair sizes with exact similarity

Stride	> 40	> 80	> 200	> 500	> 1000
500	5,569	5,569	5,569	8,799	5,377
200	89,888	89,888	89,888	8,965	5,406
120	299,933	299,933	83,940	9,113	5,441
80	640,186	640,186	108,766	9,256	5,472
40	2,440,286	931,672	178,871	12,335	5,578

Using Algorithm 7 we estimate the largest clones for exact clone detection. Unlike all other algorithms, finding the largest clones produces a set of clone pairs instead of a set of clusters. Table 7 shows that smaller window sizes generate clone pairs that combine to form larger clones. Because a data set generated using a small window size covers a smaller part of a larger clone it is expected that the combination of the smaller clone pairs will be an overestimate of the number of larger clones, but the table shows that this overestimate seems insignificant for all window sizes except window size 40.

5.5. Visualization of Clone Clusters

Figure 5 illustrates our clone detection efforts on Windows XP. Green boxes represent clones and all other colored boxes represent files within the system32 directory in Windows XP. The height of a clone (green box) represents the number of clones detected between some files. The height of a file represents the number of functions that are clones (within that file) with other functions (contained in other files), i.e. the more clones a file has, the larger the box. The width of the boxes is merely determined by each boxes' label. Different sub-directories within the system32 directory are illustrated with different colors. For instance, the *drivers* directory is color coded yellow and the *usmt* directory is colored orange.

The image reveals that much of Windows XP is somewhat related, i.e. many clones exist. However, it appears that our clone detection approach has detected correctly that most of the drivers have lots of functionality in common. The red color represents files that have the string *usr* in their name.

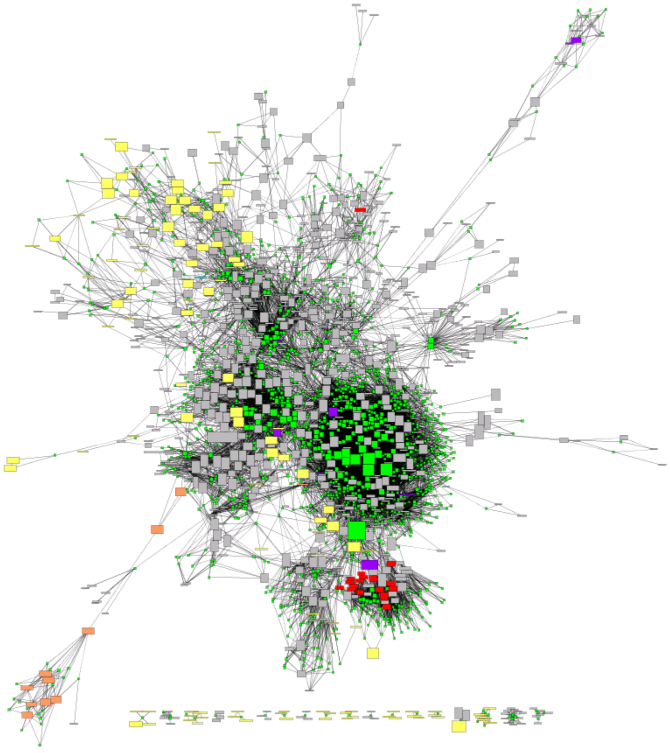

Figure 5. Clone visualization of WindowsXP

Figure 6 shows fractions of Figure 5 more in detail. For instance, *clone 37* reveals the relationship between Windows VPN components, such as *CVPNDRVA.sys*, *vpnapi.dll* and *CSGina.dll*. Similarly, *clone 166* reveals a clone relationship between the different Windows Management Instrumentation components. *Clone 986* is another clone detected by our tool that contains amongst others the Windows system information application *systeminfo*. It appears that *systeminfo* shares functionality with the system applications *tasklist*, *taskkill* and *getmac*. All the results illustrated in Figure 5 and Figure 6 use a window size of 120 for clone detection.

6. Related Work

There are no academic puclication on binary clone detection that we are aware of, but Schulman [15] did a clone study using mnemonics and API calls. This approach is not accurate since this insufficiently categorize the instruction sequence.

Many studies and tools on source clone detection and clone coverage exist. There are tools tailored toward finding plagiarism, such as Moss [14], JPlag (http://www.jplag.de). PR-Miner [13] uses frequent item set mining to detect implicit, high-level programming patterns for specification discovery and bug-detection. These tools are less accurate and therefore not applicable to source code or binary clone detection. Other tools, such as CP-Miner [12] and CCFinder [8] are available. These are token based and usually more accurate and scalable, but they tend to be sensitive to minor code changes.

Other studies include, Lague *et al.* [11], examining six versions of a telecommunications software systems; Kim *et al.* [9] investigating clones and their life span; Baxter *et al.* [4]applying AST hashing to detect exact and near-miss clones; Wahler *et al.* [17] using frequent item-set data mining techniques on ASTs to detect clones with minor changes; Basit and Jarzabek [3] using a frequent item set data mining algorithm to find design level similarities; and Jiang *et al.* [7] using characteristic vectors of ASTs to detect clones.

Our approach uses feature vectors, similar to Jiang, where we characterize normalized instruction sequences for our clone detection. However, our work is significantly different from the above studies, as we perform our clone detection on binaries and not source code. This work is novel as binary clone detection has not yet been researched.

7 Conclusions and Future Work

In this paper, we have presented a novel clone detection algorithm for binaries. We have implemented the algorithm and conducted a large-scale empirical evaluation of it on Windows XP. Results show that it is scalable and precise, thus practical to enable many applications on binaries. There are a number of possible directions for future work. First, we plan to conduct further studies with

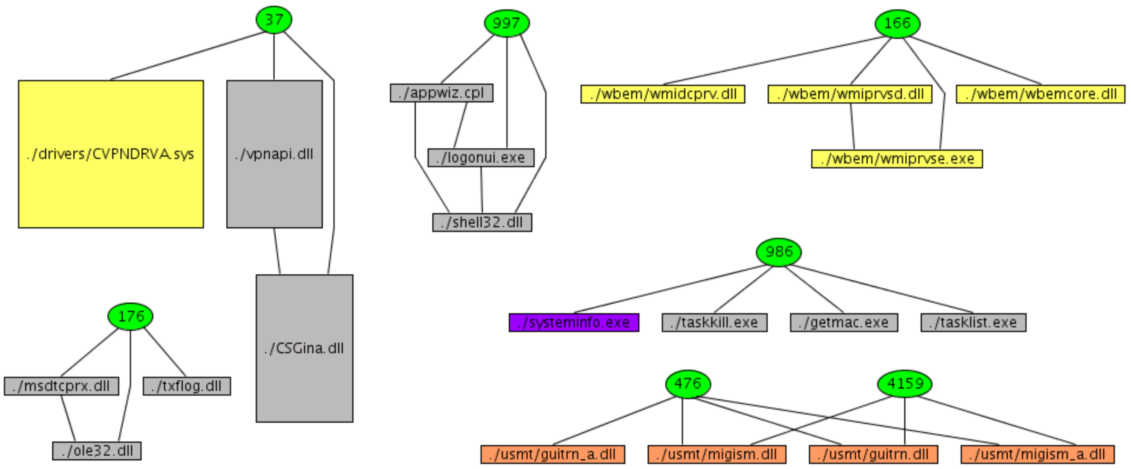

Figure 6. Visualization of selected clones in WindowsXP.

our technique, for example, by analyzing different versions of Windows and other operating systems binaries, and with other application components such as Microsoft Office. We would also like to understand the impact of different compilers and compiler optimizations on detection results. Finally, we plan to apply our technique to a number of application domains such as protecting intellectual properties and detecting latent bugs.

8 Acknowledgments

We wish to thank Lingxiao Jiang and Mark Gabel for useful insights about LSH, and helpful comments on the paper. Thanks to Taeho Kwon for Windows XP insights.

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

References

[1] A. Andoni and P. Indyk. E2LSH: Exact Euclidean locality-sensitive hashing. Web: `http://www.mit.edu/~andoni/LSH/`, 2004.

[2] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.

[3] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE-13*, pages 156–165, 2005.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[5] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[9] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[10] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[11] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, page 314, Washington, DC, USA, 1997. IEEE Computer Society.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 20–20, 2004.

[13] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, pages 306–315, 2005.

[14] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Management of Data*, pages 76–85, 2003.

[15] A. Schulman. Finding binary clones with opstrings and function digests. *Doctor Dobb's J*, 30(9):64–70, 2005.

[16] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

[17] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation*, pages 128–135, 2004.